**CS602 Term Project**

**By Sammy A. Rachman**

**Overview**

For this project, I created the foundations for an easy-to-access, easy-to-link encyclopedia site using Node.js and Mongoose/MongoDB. The site features the ability to write articles, with planned features for categorizing and author-management. Important functionality for this site includes the capability to acquire and import new articles using XML format as well as the nearly completed feature for linking between articles using text tags. The web application is focused primarily on back-end performance, with coding practices, design decisions and organization aiming to improve scalability, flexibility and well-defined categories for scripts.

**Start-up Guide**

1. Run the mongoDB server – credentials can be changed in ‘/credentials.js’.
2. Run npm install to install packages available in ‘/packages.json’.
3. Run node client.js to prepopulate the database. Exit when the script completes.
4. Run node server.js to initialize the server.
5. Browse to localhost:3000 to access the site.

**Use Guide**

1. All RESTful/CRUD features are implemented for both Articles and Authors (list, show, add/create, edit/update, delete). ShowXML and Upload XML feature for Article is also included.
2. Creating a new article with a new author name that does not exist in the Author collection will automatically create that author.
3. ShowXML for an article will display the complete XML for that article; a more in-depth XML import guide is viewable below.
4. When writing the introduction, content or conclusion of an article, encapsulate names of other articles with square brackets ([[ ]] -- e.g. ‘[[ArticleName]]‘) to generate a link to that other article when viewing the article. *NOTE: Only one link can be used in an article at this time.*

**XML Guide**

When uploading a new article via XML, use the format below:

<article>

<title> titlename </title>

<introduction> introduction </introduction>

<contents> contents </contents>

<conclusion> conclusion </conclusion>

<category> category </category>

<author> authorName </author>

</article>

**Design Choices**

The web app was designed with back-end performance and organization of the MVC in mind. As a result, the project tree (seen next page) segregates functionality in an understandable manner. Module exports and requirements were set up in a way to keep ownership for functionality and access to models to those that are directly responsible.

Therefore, for example, only instances of Authors should have direct access to authorController functions. Any interaction with Articles (such as requiring a list of all articles written by the author) will go through the auhtorArticleController.

* Models – contains schema and connection details
* Controllers – contains routing and functionality for instances of models
* Views – Contains all views and styling.

In addition, the following npm modules were used:

* Body-parser
* Express
* Express-handlebars
* Mongoose
* Xml2js
* StringJS

**Linking Article and Author models**

The Article schema contains a reference to an author object in the \_author field. This is intentional to allow a reference (akin to a relational DB) to an existing author. This creates the necessity of implementing a findOrSaveAuthor functionality (seen in authorArticleController.js), where an author is queried. If found, the author will be returned. Otherwise, the author will be created. The author’s \_id will be linked in the article’s \_author field. This utilizes asynchronous DB functions, so the importance of utilizing returns and promise chains were needed. Also of note is utilizing the .populate() function for passing in an author’s name in situations where article is queried (such as in article.show).

**XML Functionality**

The XML capability was added primarily as a way to quickly add articles as content became available down the line for a site. The actual tags and schema of an article can be renamed in the future, but the functionality is very useful for the purposes of adding articles that are already written externally from the website (e.g. written from a .doc file, once tags are added appropriately they can be pasted into the Upload XML feature).

As styling is handled by the web application and not on the user, the tags – which directly correlate to the Article schema (which can change) would allow the web developer to set up article displays and formats in a way they want, and XML tags will denote those sections. For example, currently in the site’s showArticleView the Introduction, Content and Conclusion are all separate divs, so they can all be styled and any article being imported via XML or added manually will follow the site’s styling for each of those divs.

**[Work-in-Progress] Article-linking**

The other core function for the website is the article-linking capability. Unfortunately, while it is functional it is not yet completed. At this time, only one link can be put in a single record (introduction, contents, conclusion) at a time – any more will still cause errors. This can be fixed but was unable to be solved before the submission deadline. Utilizing RegEx to detect text with sharp brackets in the article’s contents, it will prompt the system to look up an article with the item’s name and replace it with an HTML link to be passed into the handlebars data. This happens as the user navigates into the showArticle page.

This was a very challenging implementation as there are multiple asynchronous queries being made in the promise chain, and understanding where a return or a Promise.resolve() for synchronous results is a critical component for completing this functionality.

**Future Plans**

In the future, important implementations are necessary. Of course, the WIP article-linking functionality should be completed first. Afterwards, the following could be added into the web application:

1. **Secure Inputs and Validation**

Right now, inputs and validations are currently minimal. In the future, all inputs should be sanitized and validated appropriately. Strings should be checked and sanitized, white space should be handled, and so on. StringJS, used in this project can be used to handle whitespace.

1. **User and Admin Functionality**

Right now, there is nothing to stop any user connecting from creating/deleting/editing articles or authors. With a user/admin functionality, features such as publishing articles, hiding articles, requiring sign-in to be a listed author, etc. are capable. Passport seems to be a popular module to implement this functionality.

**Project Tree:**

* Root
  + client.js
  + credentials.js
  + package.json
  + server.js
  + **controllers**
    - articleController.js
    - authorArticleController.js
    - authorController.js
    - index.js
    - *show script.js*
    - xmlController.js
  + **models**
    - article.js
    - author.js
    - dbConnection.js
  + **views**
    - **layouts**
      * main.handlebars
      * styles.css
    - 404.handlebars
    - addArticleView.handlebars
    - addAuthorView.handlebars
    - editArticleView.handlebars
    - editAuthorView.handlebars
    - listArticlesView.handlebars
    - listAuthorsView.handlebars
    - showArticleView.handlebars
    - showAuthorView.handlebars
    - xmlArticleView.handlebars